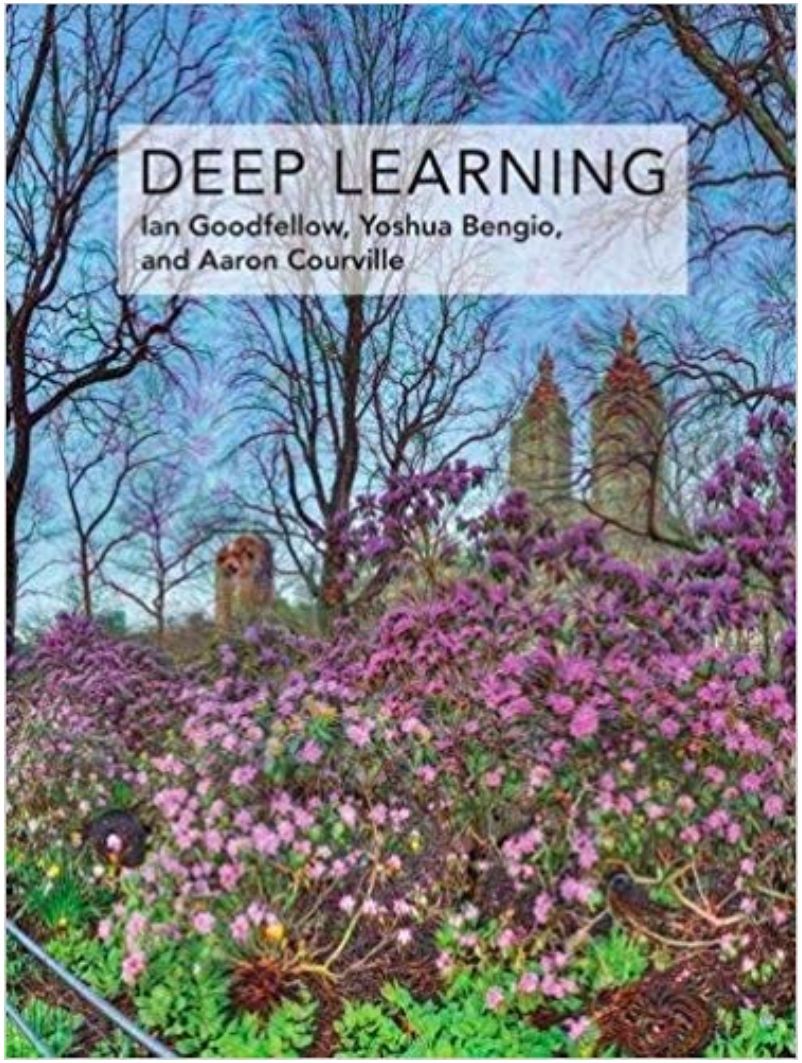
國立陽明交通大學
NATIONAL YANG MING CHIAO TUNG UNIVERSITY

Deep Learning
深度學習
Fall 2025

Optimization for Training
(Chapter 8.3-8.5)

Prof. Chia-Han Lee
李佳翰 教授



- Figure source: Textbook and Internet
- You are encouraged to buy the textbook.
- Please respect the copyright of the textbook. Do not distribute the materials to other people.

Stochastic gradient decent

- A crucial parameter for the stochastic gradient descent (SGD) algorithm is the learning rate.

- In practice, it is necessary to gradually decrease the learning rate over time. This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not vanish even when we arrive at a minimum.

- By comparison, the true gradient of the total cost function becomes small and then $\mathbf{0}$ when we approach and reach a minimum using batch gradient descent, so batch gradient descent can use a fixed learning rate.

Stochastic gradient decent

- Denote the learning rate at iteration k as ϵ_k. A sufficient condition to guarantee convergence of SGD is:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \quad \text{and} \tag{8.12}$$

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty. \tag{8.13}$$

- In practice, it is common to decay the learning rate linearly until iteration τ:

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau \tag{8.14}$$

with $\alpha = \frac{k}{\tau}$. After iteration τ, we leave ϵ a constant.

- The learning rate may be chosen by trial and error, but it is usually best to choose it by monitoring learning curves that plot the objective function as a function of time.

Stochastic gradient decent

- Usually τ may be set to the number of iterations required to make a few hundred passes through the training set. Usually ϵ_τ is set to roughly 1 percent the value of ϵ_0.

- The main question is how to set ϵ_0. If it is too large, the learning curve will show violent oscillations, with the cost function often increasing significantly. If the learning rate is too low, learning proceeds slowly, and if the initial learning rate is too low, learning may become stuck with a high cost value.

- Typically, the optimal initial learning rate is higher than the learning rate that yields the best performance after the first 100 iterations or so. Therefore, it is usually best to monitor the first several iterations and use a learning rate that is higher than the best-performing one.

Stochastic gradient decent

- To study the convergence rate of an optimization algorithm it is common to measure the excess error $J(\boldsymbol{\theta}) - \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, which is the amount by which the current cost function exceeds the minimum possible cost.

- When SGD is applied to a convex problem, the excess error is $O(1/\sqrt{k})$ after k iterations, while in the strongly convex case, it is $O(1/k)$.

- Batch gradient descent enjoys better convergence rates than stochastic gradient descent in theory. However, the Cramér-Rao bound states that generalization error cannot decrease faster than $O(1/k)$. It therefore may not be worthwhile to pursue an optimization algorithm that converges faster than $O(1/k)$.

Stochastic gradient decent

- With large datasets, the ability of SGD to make rapid initial progress while evaluating the gradient for very few examples outweighs its slow asymptotic convergence.

- Most of the algorithms described in the remainder of this chapter achieve benefits that matter in practice but are lost in the constant factors obscured by the $O(1/k)$ asymptotic analysis.

- One can also trade off the benefits of both batch and stochastic gradient descent by gradually increasing the minibatch size during the course of learning.

Momentum

- While stochastic gradient descent remains a popular optimization strategy, learning with it can sometimes be slow.

- The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.

- The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

Momentum

- Formally, the momentum algorithm introduces a variable v that plays the role of velocity—it is the direction and speed at which the parameters move through parameter space.

- The velocity is set to an exponentially decaying average of the negative gradient. The name momentum derives from a physical analogy, in which the negative gradient is a force moving a particle through parameter space, according to Newton's laws of motion.

- Momentum in physics is mass times velocity. In the momentum learning algorithm, we assume unit mass, so the velocity vector v may also be regarded as the momentum of the particle.

Momentum

- A hyperparameter $\alpha \in [0, 1)$ determines how quickly the contributions of previous gradients exponentially decay.

- The update rule is given by

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right), \tag{8.15}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}. \tag{8.16}$$

  The velocity $\boldsymbol{v}$ accumulates the gradient elements $\nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$.

- The larger α is relative to ϵ, the more previous gradients affect the current direction.

Momentum

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

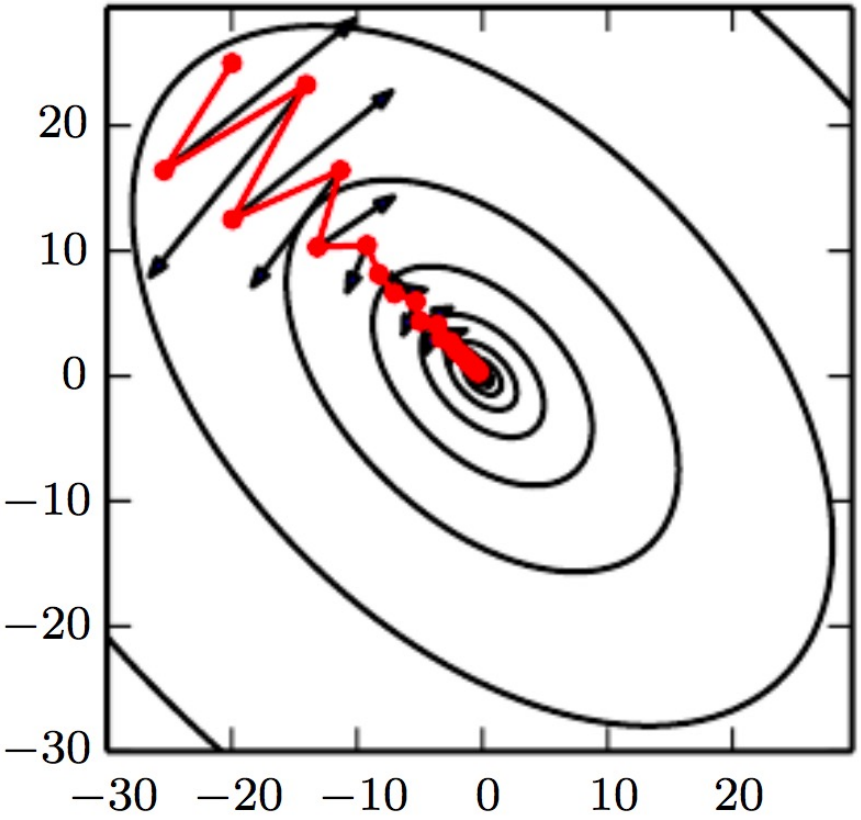
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

end while

Momentum

The red path cutting across the contours indicates the path followed by the momentum learning rule as it minimizes this function. Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon.



Momentum

- The size of the step depends on how large and how aligned a sequence of gradients are.
- The step size is largest when many successive gradients point in exactly the same direction. If the momentum algorithm always observes gradient $\boldsymbol{g}$, then it will accelerate in the direction of $-\boldsymbol{g}$, until reaching a terminal velocity where the size of each step is $\frac{\epsilon\|\boldsymbol{g}\|}{1-\alpha}$.
- It is thus helpful to think of the momentum hyperparameter in terms of $\frac{1}{1-\alpha}$. For example, $\alpha = 0.9$ corresponds to multiplying the maximum speed by 10 relative to the gradient descent algorithm.

Momentum

- Common values of α used in practice include 0.5, 0.9, and 0.99. Like the learning rate, α may also be adapted over time. Typically it begins with a small value and is later raised.

- Adapting α over time is less important than shrinking ϵ over time.

- We can view the momentum algorithm as simulating a particle subject to continuous-time Newtonian dynamics. The physical analogy can help build intuition for how the momentum and gradient descent algorithms behave.

Momentum

- The position of the particle at any point in time is given by $\boldsymbol{\theta}(t)$. The particle experiences net force $\boldsymbol{f}(t)$. This force causes the particle to accelerate:

$$\boldsymbol{f}(t) = \frac{\partial^2}{\partial t^2}\boldsymbol{\theta}(t). \tag{8.18}$$

- Rather than viewing this as a second-order differential equation of the position, we can introduce the variable $\boldsymbol{v}(t)$ representing the velocity of the particle at time t and rewrite the Newtonian dynamics as a first-order differential equation:

$$\boldsymbol{v}(t) = \frac{\partial}{\partial t}\boldsymbol{\theta}(t), \tag{8.19}$$

$$\boldsymbol{f}(t) = \frac{\partial}{\partial t}\boldsymbol{v}(t). \tag{8.20}$$

Momentum

- The momentum algorithm then consists of solving the differential equations via numerical simulation.

- A simple numerical method for solving differential equations is Euler's method, which simply consists of simulating the dynamics defined by the equation by taking small, finite steps in the direction of each gradient. This explains the basic form of the momentum update.

- What specifically are the forces? One force is proportional to the negative gradient of the cost function: $-\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. This force pushes the particle downhill along the cost function surface.

Momentum

- The Newtonian scenario used by the momentum algorithm uses this force to alter the velocity of the particle. Whenever it descends a steep part of the surface, it gathers speed and continues sliding in that direction until it begins to go uphill again.

- One other force is necessary. If the only force is the gradient of the cost function, then the particle might never come to rest. To resolve this problem, we add one other force, proportional to $-\boldsymbol{v}(t)$.

- In physics terminology, this force corresponds to viscous drag, as if the particle must push through a resistant medium such as syrup. This causes the particle to gradually lose energy over time and eventually converge to a local minimum.

Momentum

- Why do we use $-\boldsymbol{v}(t)$ and viscous drag in particular?
- Part of the reason to use $-\boldsymbol{v}(t)$ is mathematical convenience—an integer power of the velocity is easy to work with.
- Yet other physical systems have other kinds of drag based on other integer powers of the velocity. For example, a particle traveling through the air experiences turbulent drag, with force proportional to the square of the velocity, while a particle moving along the ground experiences dry friction, with a force of constant magnitude.
- We can reject each of these options.

Momentum

- Turbulent drag becomes very weak when the velocity is small. It is not powerful enough to force the particle to come to rest. A particle with a nonzero initial velocity that experiences only the force of turbulent drag will move away from its initial position forever.

- If we use a power of zero, representing dry friction, then the force is too strong. When the force due to the gradient of the cost function is small but nonzero, the constant force due to friction can cause the particle to come to rest before reaching a local minimum.

- Viscous drag is weak enough that the gradient can continue to cause motion until a minimum is reached, but strong enough to prevent motion if the gradient does not justify moving.

Nesterov momentum

- A variant of the momentum algorithm is inspired by Nesterov's accelerated gradient method. The update rules in this case are given by

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_{i=1}^{m} L\left(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(i)} \right) \right], \tag{8.21}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}, \tag{8.22}$$

  where the parameters α and ϵ play a similar role as in the standard momentum method.

- The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum, the gradient is evaluated after the current velocity is applied. Thus one can interpret Nesterov momentum as attempting to add a correction factor to the standard method of momentum.

Nesterov momentum

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.

Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$.

Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

end while

Nesterov momentum

- In the convex batch gradient case, Nesterov momentum brings the rate of convergence of the excess error from $O(1/k)$ (after k steps) to $O(1/k^2)$.

- Unfortunately, in the stochastic gradient case, Nesterov momentum does not improve the rate of convergence.

Algorithms with adaptive learning rates

- The momentum algorithm can somewhat mitigate the difficulty of setting the learning rate, but it does so at the expense of introducing another hyperparameter. Is there another way?

- If we believe that the directions of sensitivity are somewhat axis aligned, it can make sense to use a separate learning rate for each parameter and automatically adapt these learning rates throughout the course of learning.

AdaGrad

- The AdaGrad algorithm individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient.

- The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate.

- The net effect is greater progress in the more gently sloped directions of parameter space.

AdaGrad

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.

Compute update: $\Delta \boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.

end while

AdaGrad

- In the context of convex optimization, the AdaGrad algorithm enjoys some desirable theoretical properties, and converge rapidly.

- Empirically, however, for training deep neural network models, the accumulation of squared gradients from the beginning of training can result in a premature and excessive decrease in the effective learning rate. It may have made the learning rate too small before arriving at a locally convex bowl.

- AdaGrad performs well for some but not all deep learning models.

RMSProp

- The RMSProp algorithm performs better in the nonconvex setting by changing the gradient accumulation into an exponentially weighted moving average.

- RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of AdaGrad initialized within that bowl.

- Compared to AdaGrad, the use of the moving average introduces a new hyperparameter, ρ, that controls the length scale of the moving average.

- Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for DNNs.

RMSProp

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers

Initialize accumulation variables $\boldsymbol{r} = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.

Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$.

end while

RMSProp

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$

Initialize accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.

Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

end while

Adam

- Adam is yet another adaptive learning rate optimization algorithm. The name "Adam" derives from the phrase "adaptive moments."

- It is best seen as a variant on the combination of RMSProp and momentum with a few important distinctions.

- First, in Adam, momentum is incorporated directly as an estimate of the first-order moment (with exponential weighting) of the gradient. The most straightforward way to add momentum to RMSProp is to apply momentum to the rescaled gradients. The use of momentum in combination with rescaling does not have a clear theoretical motivation.

Adam

- Second, Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin.

- RMSProp also incorporates an estimate of the (uncentered) second-order moment; however, it lacks the correction factor. Thus, unlike in Adam, the RMSProp second-order moment estimate may have high bias early in training.

- Adam is generally regarded as being fairly robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default.

Adam

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)

Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)

Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})

Require: Initial parameters $\boldsymbol{\theta}$

Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$

Initialize time step $t = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

$t \leftarrow t + 1$

Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$

Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$

Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$

Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$

Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

Choosing the right optimization algorithm

- Which algorithm should one choose?
- Unfortunately, there is currently no consensus on this point.
- Currently, the most popular optimization algorithms actively in use include SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta, and Adam.
- The choice of which algorithm to use, at this point, seems to depend largely on the user's familiarity with the algorithm (for ease of hyperparameter tuning).

Parameter initialization strategies

- Most algorithms are strongly affected by the choice of initialization.
- The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether.
- When learning does converge, the initial point can determine how quickly learning converges and whether it converges to a point with high or low cost. Also, points of comparable cost can have wildly varying generalization error, and the initial point can affect the generalization.

Parameter initialization strategies

- Modern initialization strategies are simple and heuristic. Designing improved initialization strategies is a difficult task because NN optimization is not yet well understood.

- Most initialization strategies are based on achieving some nice properties when the network is initialized. However, we do not have a good understanding of which of these properties are preserved under which circumstances after learning begins to proceed.

- A further difficulty is that some initial points may be beneficial from the viewpoint of optimization but detrimental from the viewpoint of generalization. Our understanding of how the initial point affects generalization is especially primitive.

Parameter initialization strategies

- Perhaps the only property known with complete certainty is that the initial parameters need to "break symmetry" between different units.

- If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters. If they have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way.

Parameter initialization strategies

- Even if the model or training algorithm is capable of using stochasticity to compute different updates for different units, it is usually best to initialize each unit to compute a different function from all the other units.

- This may make sure that no input patterns are lost in the null space of forward propagation and that no gradient patterns are lost in the null space of back-propagation.

- The goal of having each unit compute a different function motivates random initialization of the parameters. Random initialization from a high-entropy distribution over a high-dimensional space is computationally cheaper and unlikely to assign any units to compute the same function as each other.

Parameter initialization strategies

- We could explicitly search for a large set of basis functions that are all mutually different from each other using such as Gram-Schmidt orthogonalization, but this often incurs a noticeable computational cost.

- Typically, we set the biases for each unit to heuristically chosen constants, and initialize the weights randomly.

- We almost always initialize all the weights in the model to values drawn randomly from a Gaussian or uniform distribution. The choice of Gaussian or uniform distribution does not seem to matter much. The scale of the initial distribution, however, does have a large effect on both the outcome of the optimization procedure and the ability of the network to generalize.

Parameter initialization strategies

- Larger initial weights will yield a stronger symmetry-breaking effect, helping to avoid redundant units. They also help to avoid losing signal during forward or back-propagation through the linear component of each layer.

- Initial weights that are too large may, however, result in exploding values during forward propagation or back-propagation.

- In recurrent networks, large weights can also result in chaos (such extreme sensitivity to small perturbations of the input that the behavior of the deterministic forward propagation procedure appears random).

Parameter initialization strategies

- To some extent, the exploding gradient problem can be mitigated by gradient clipping (thresholding the values of the gradients before performing a gradient descent step).
- Large weights may also result in extreme values that cause the activation function to saturate, causing complete loss of gradient through saturated units.
- These competing factors determine the ideal initial scale of the weights.

Parameter initialization strategies

- The perspectives of regularization and optimization can give very different insights into how we should initialize a network.

- The optimization perspective suggests that the weights should be large enough to propagate information successfully, but some regularization concerns encourage making them smaller.

- The use of an optimization algorithm, such as stochastic gradient descent, that makes small incremental changes to the weights and tends to halt in areas that are nearer to the initial parameters expresses a prior that the final parameters should be close to the initial parameters.

Parameter initialization strategies

- Some heuristics are available for choosing the initial scale of the weights. One heuristic is to initialize the weights of a fully connected layer with m inputs and n outputs by sampling each weight from $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$, while another uses the normalized initialization

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right). \tag{8.23}$$

- This latter heuristic is designed to compromise between the goal of initializing all layers to have the same activation variance and the goal of initializing all layers to have the same gradient variance.

Parameter initialization strategies

- It was recommend initializing to random orthogonal matrices, with a carefully chosen scaling or gain factor g that accounts for the nonlinearity applied at each layer.

- It was later shown that setting the gain factor correctly is sufficient to train networks as deep as 1,000 layers, without needing to use orthogonal initializations.

- A key insight of this approach is that in feedforward networks, activations and gradients can grow or shrink on each step of forward or back-propagation, following a random walk behavior. This is because feedforward networks use a different weight matrix at each layer. If this random walk is tuned to preserve norms, then feedforward networks can mostly avoid the vanishing and exploding gradients problem.

Parameter initialization strategies

- In practice, we usually need to treat the scale of the weights as a hyperparameter whose optimal value lies somewhere roughly near but not exactly equal to the theoretical predictions.

- This may be for three different reasons:
  - We may be using the wrong criteria—it may not actually be beneficial to preserve the norm of a signal throughout the entire network.
  - Second, the properties imposed at initialization may not persist after learning has begun to proceed.
  - Third, the criteria might succeed at improving the speed of optimization but inadvertently increase generalization error.

Parameter initialization strategies

- One drawback to scaling rules that set all the initial weights to have the same standard deviation, such as $\frac{1}{\sqrt{m}}$, is that every individual weight becomes extremely small when the layers become large.

- In an alternative initialization scheme, called sparse initialization, each unit is initialized to have exactly k nonzero weights. The idea is to keep the total amount of input to the unit independent from the number of inputs m without making the magnitude of individual weight elements shrink with m.

Parameter initialization strategies

- Sparse initialization helps to achieve more diversity among the units at initialization time. However, it also imposes a very strong prior on the weights that are chosen to have large Gaussian values.

- Because it takes a long time for gradient descent to shrink “incorrect” large values, this initialization scheme can cause problems for units, such as maxout units, that have several filters that must be carefully coordinated with each other.

Parameter initialization strategies

- When computational resources allow it, it is usually a good idea to treat the initial scale of the weights for each layer as a hyperparameter, and to choose these scales using a hyperparameter search algorithm.

- Alternately, one can manually search for the best initial scales. A good rule of thumb for choosing the initial scales is to look at the range or standard deviation of activations or gradients on a single minibatch of data. This procedure can in principle be automated and is generally less computationally costly than hyperparameter optimization based on validation set error because it is based on feedback from the behavior of the initial model on a single batch of data.

Parameter initialization strategies

- Initialization of other parameters is typically easier.

- The approach for setting the biases must be coordinated with the approach for setting the weights. Setting the biases to zero is compatible with most weight initialization schemes.

There are a few situations where we may set some biases to nonzero values:

1. If a bias is for an output unit, then it is often beneficial to initialize the bias to obtain the right marginal statistics of the output. To do this, we assume that the initial weights are small enough that the output of the unit is determined only by the bias.

Parameter initialization strategies

- This justifies setting the bias to the inverse of the activation function applied to the marginal statistics of the output in the training set.

- For example, if the output is a distribution over classes, and this distribution is a highly skewed distribution with the marginal probability of class i given by element c_i of some vector $\boldsymbol{c}$, then we can set the bias vector $\boldsymbol{b}$ by solving the equation $\mathrm{softmax}(\boldsymbol{b}) = \boldsymbol{c}$.

- This applies also to models such as autoencoders and Boltzmann machines. These models have layers whose output should resemble the input data $\boldsymbol{x}$, and it can be very helpful to initialize the biases of such layers to match the marginal distribution over $\boldsymbol{x}$.

Parameter initialization strategies

2. Sometimes we may want to choose the bias to avoid causing too much saturation at initialization.

- For example, we may set the bias of a ReLU hidden unit to 0.1 rather than 0 to avoid saturating the ReLU at initialization.

- This approach is not compatible with weight initialization schemes that do not expect strong input from the biases. For example, it is not recommended for use with random walk initialization.

Parameter initialization strategies

3. Sometimes a unit controls whether other units are able to participate in a function.

- In such situations, we have a unit with output u and another unit $h \in [0,1]$, and they are multiplied together to produce an output uh. We can view h as a gate that determines whether $uh \approx u$ or $uh \approx 0$.

- In these situations, we want to set the bias for h so that $h \approx 1$ most of the time at initialization. Otherwise u does not have a chance to learn.

- For example, we may set the bias to 1 for the forget gate of the LSTM model.

Parameter initialization strategies

- Another common type of parameter is a variance or precision parameter. For example, we can perform linear regression with a conditional variance estimate

$$p(y \mid \boldsymbol{x}) = \mathcal{N}(y \mid \boldsymbol{w}^T \boldsymbol{x} + b, 1/\beta), \tag{8.24}$$

where β is a precision parameter.

- We can usually initialize variance or precision parameters to 1 safely.

Parameter initialization strategies

- Besides these simple constant or random methods of initializing model parameters, it is possible to initialize model parameters using machine learning.
- A common strategy is to initialize a supervised model with the parameters learned by an unsupervised model trained on the same inputs.
- One can also perform supervised training on a related task.
- Even performing supervised training on an unrelated task can sometimes yield an initialization that offers faster convergence than a random initialization.

Parameter initialization strategies

- Some of these initialization strategies may yield faster convergence and better generalization because they encode information about the distribution in the initial parameters of the model.

- Others apparently perform well primarily because they set the parameters to have the right scale or set different units to compute different functions from each other.